LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-ID-155540

CALE Implementation Guide

J. Hagelberg

September 1, 2003

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Table of Contents

Overview

During the summer of 2003, a new version of CALE was written which uses wxWindows, a platform-independent graphics library that also provides support for threads, file handling, and many other things. When the project was started, there were several different approaches considered. The first approach that was considered was to create a version of CALE using the Windows API. This seemed like a logical choice since the primary goal of the project was to create a version of CALE that would run on Windows. However, if the graphics used in CALE were going to be completely rewritten, it seemed to make more sense to do the new graphics programming using a platform independent library. This way we would be able to get rid of the several different versions of CALE that exist for various platforms and instead just use this new version on all of the platforms. Out of all of the platform independent graphics packages that were considered, wxWindows was chosen because it is open source, robust, and provides support for drawing low-level graphics primitives like lines and polygons.

Even though the original CALE code was written in C, the new parts are written in C++. The primary reason for this is that wxWindows is written in C++. C++ also makes it much easier to design components that can be easily reused in other projects and supports polymorphism and templates.

Many places in this document, you will see `$(WXDIR)`. This is a macro that was defined in the makefile and contains the wxWindows directory. For the environment used in development, it was defined as `c:/wxWindows-2.4.1`.

This document provides more details about how the new version of CALE was created and the reasoning behind them.

Development Environment

The code for this project was developed on a machine running Windows 2000 Service Pack 3. The compiler used was MinGW using gcc version 3.2.3. The code was tested using the debug version of wxWindows 2.4.1 for Windows.

Prerequisites

In order to use the wxWindows version of CALE, you must have wxWindows installed. wxWindows has been implemented for every major platform and can be downloaded from http://www.wxwindows.org/. Additionally, ImageMagick is required if you want to be able to create GIF, JPEG, or EPS files. ImageMagick is also open source and can be downloaded from http://www.imagemagick.org/.

Compilation

To debug or to not debug

When compiling the wxWindows version of CALE, one needs to consider the question of whether or not to include debug symbols in the executable. Furthermore, one must also consider whether or not wxWindows was compiled in its own debug mode using the `__WXDEBUG__` flag. The latter is an issue because wxWindows will crash if the debug and non-debug versions are mixed together. If wxWindows was compiled in its debug mode, then the source code must also be compiled using debug mode by defining `__WXDEBUG__` during compilation.

If you are unsure whether or not wxWindows was compiled using its own debug mode, there is a fairly simple way to find out. When wxWindows is compiled using `__WXDEBUG__`, a "d" is appended to the end of its library names so that they have names like libwxbased.a. The non-debug version of this particular file is named libwxbase.a. The libraries are usually located in a subdirectory called "lib" within the wxWindows directory.

Now that you know how to use wxWindows in its own debug mode, you might wonder what its advantages are. The main advantage is that defining `__WXDEBUG__` enables many run-time checks such as array bounds checking for `wxArrays`. These can be very useful when debugging code. In the release version of CALE, of course, wxWindows should not be used in its debug mode.

As you might imagine, whether or not `__WXDEBUG__` is defined is completely unrelated to whether or not debugging symbols are included in the executable. To include them when using gcc, use the –g flag. Other compilers may use different flags.

Compiler Flags

There are a number of definitions, includes, libraries, and other flags that that need to be present so that the wxWindows header files know what compiler and platform is being used. The easiest way to find out what compiler flags you need is to compile one of the wxWindows samples using the makefile they provide and then look at what definitions, flags, includes, and so forth are required. Here is a listing of the ones used during development. They are only meant to be a guide as to what compiler flags are needed. Most of the flags here should work with any program compiled in a similar development environment. Any flags that are specific to CALE have been indicated.

Compiler Definitions

```
__WXMSW__, WINVER=0x0400, __WIN95__, __GNUWIN32__, STRICT,
HAVE_W32API_H, WINDOWS, __WXDEBUG__, WXWIN, IMAGEMAGICK,
UNIX, OPENHE, LITEND, IEEE64
```

The definitions `WXWIN, IMAGEMAGICK, UNIX, OPENHE, LITEND, IEEE64` are all specific to CALE. The `WXWIN` definition tells the code that is should use wxWindows. See the makefile for a complete description of the other flags.

Compiler Include Directories

```
$(WXDIR)/lib/mswd, $(WXDIR)/include,
$(WXDIR)/contrib/include, $(WXDIR)/src/regex,
$(WXDIR)/src/png, $(WXDIR)/src/jpeg, $(WXDIR)/src/zlib,
$(WXDIR)/src/tiff
```

Other Compiler Flags

```
-x c++ -fno-implicit-templates
```

These flags are specific to CALE. The first flag (`-x C++`) tells gcc that all input files should be treated as C++. The second flag (`-fno-implicit-templates`) tells the compiler to only instantiate the templates that it is explicitly told to instantiate in the code. Your compiler may require different flags to do these things.

Library Directories for Linker

```
$(WXDIR)/lib, $(WXDIR)/contrib/lib
```

Libraries for Linker

```
m, stdc++, gcc, odbc32, wsock32, winspool, winmm, shell32,
comctl32, ctl3d32, advapi32, ole32, loleaut32, uuid
```

Note that the wxWindows library is not included here. Whether or not you need to include the wxWindows library here depends on how you compiled wxWindows. If you compiled it as a shared library, include it. During development, the wxWindows library was treated as just another object file. See the example linker command for how this is done.

The only library here that is specific to CALE is "m."

Other Linker Flags

```
-Wl,--subsystem,console -mwindows
```

These flags tell the linker that we need to have a console window and that we are using windows.

Resource Files

When a person writes a program targeted at the Windows platform, resource files are generally used to store things like bitmaps, icons, accelerator tables, and other such things. The same is true for the wxWindows version of CALE for Windows. When CALE is being compiled on the Windows platform, the CALE resource file, cale.rc, must be compiled into an object file that can then be linked with the other compiled libraries. This can be done with the following command using windres.exe, which comes with MinGW:

```
windres.exe --use-temp-file --include-dir $(WXDIR)/include
--define __WIN32__ --define __WIN95__ --define __GNUWIN32__
-i cale.rc -o cale_resources.o
```

Examples

Example compiler invocation:

```
gcc -I$(WXDIR)/lib/mswd -I$(WXDIR)/include -
I$(WXDIR)/contrib/include -I$(WXDIR)/src/regex -
```

```
I$(WXDIR)/src/png -I$(WXDIR)/src/jpeg -I$(WXDIR)/src/zlib -
I$(WXDIR)/src/tiff -DUNIX -DOPENHE -DLITEND -DIEEE64 -
DWXWIN -DIMAGEMAGICK -DWINVER=0x0400 -D__WIN95__ -
D__GNUWIN32__ -DSTRICT -DHAVE_W32API_H -D__WXMSW__ -
D__WINDOWS__-D__WXDEBUG__ -x c++ -g -fno-implicit-templates
-c ActionList.cpp
```

Example linker invocation:

```
gcc -g -o cale.exe {all object files} -Wl,--
subsystem,console -mwindows -L$(WXDIR)/lib -
L$(WXDIR)/contrib/lib $(WXDIR)/lib/libwxmswd.a -lm -lstdc++
-lgcc -lodbc32 -lwsock32 -lwinspool -lwinmm -lshell32 -
lcomctl32 -lctl3d32 -ladvapi32 -lole32 -loleaut32 -luuid
```

Implementation Details

Naming Conventions

In implementing CALE for wxWindows, the following naming conventions were used:

- Class variables are prefixed with `m_`
- Static class variables are prefixed with `sm_`
- `#defined` macros are in upper case
- All class names are in title case
- Function names are in title case with the exception of the first letter, which is in lower case
- For `wxBoxSizers`: suffix is `_h` is it is laid out horizontally and `_v` if it is laid out vertically

Motivation

When considering how to implement the wxWindows version of CALE, there were several goals. The first goal was to keep CALE and the new graphics code as separate as possible in order to minimize the coupling between the two. Another goal was to create the new version using components that for the most part could be easily reused in other parts of the code and in other projects. In order to make the code more maintainable, code duplication was avoided whenever it was possible. An additional goal, especially when it came to redesigning the Console, was to make the user interface as easy to use as possible.

Implementation Overview

In the wxWindows version of CALE, there are always two threads: one for reading and executing commands and one for the graphics. There are several reasons that two threads are necessary. To begin with, the functions that read input from the keyboard are by necessity blocking functions. They don't return until Enter has been pressed. If the command line interaction took place in the same thread as the graphics window, the graphics window would be completely unresponsive, and that is not acceptable. As a practical matter, the original CALE code needs to be running in its own thread. This is because integrating everything into one thread would involve rewriting the event-handling loop in wxWindows to periodically do something related to CALE like reading input from the command window.

Design patterns such as the observer pattern are used extensively in the wxWindows version of CALE, especially in the Console dialog boxes. These patterns are used to minimize coupling between classes and are discussed in detail later on.

In order to keep open the option of compiling the code without wxWindows, all of the wxWindows specific code in the original .c files is enclosed by an `#ifdef WXWIN` directive.

As of this writing, the following original CALE files have been modified: ccmd.c, cctl.c, ccur.c, cflash.c, cgl.c, cnsl.c, cparm.c, cpix.c, and cplot.c.

Synchronization

One consequence of there always being two threads present is that there is now the possibility that two commands could be executing simultaneously: one executed from the Console and one executed from the command line. In order to prevent this from happening, it is necessary to lock a mutex at the beginning and unlock a mutex at the end of all functions that are called by the Console. As of this writing, this has not been done. This issue is discussed more fully in the Future Work section. Another potential synchronization issue can occur if CALE is trying to do drawing while the Graphics Window is being repainted. This is because the drawing routines are not thread-safe. This issue was resolved by locking a mutex before doing drawing calls and unlocking it afterwards. Additionally, automatic repainting is disabled when CALE is executing a “run” command.

How Drawing Works

One of the primary goals in implementing the actual drawing in CALE was to keep the drawing code and the original CALE code as separate as possible. In order to this, an abstraction layer was created between CALE and the native wxWindows drawing commands. The abstraction layer allows for greater code reuse by abstracting out potentially complex drawing and allowing it to take place using a single function call. This abstraction is achieved by creating one class for each drawing event that occured. For example, there is a `Rectangle` class that knows how to draw a rectangle onto a device context, a `FilledRectangle` class that knows how to draw a filled rectangle onto a device context, and a `FontSizeChanger` class which knows how to change the font size inside a device context. All of these classes are subclasses of a base class called `GraphicsEvent`. Not all of the GraphicsEvent classes correspond directly to a low-level drawing call. In fact, the drawing they do could be arbitrarily complex. That is what makes the abstraction layer so nice. It hides the details about how the drawing is actually accomplished and thus simplifies the task of implementing the drawing while at the same time making it less error-prone.

In order to make the drawing actually happen, some graphics event, for example a `FontSizeChanger`, is instantiated in the original CALE code and then passed to the

DrawingFrame using either its apply or draw method (they are identical – I have been using the convention that you draw anything that can be seen and you apply everything else). This graphics event is eventually passed to the CaleDeviceContext, which then instructs the graphics event to do some sort of drawing onto it. In the case of the FontSizeChanger, the graphics event would change the font size within the CaleDeviceContext. The following diagram illustrates this process.

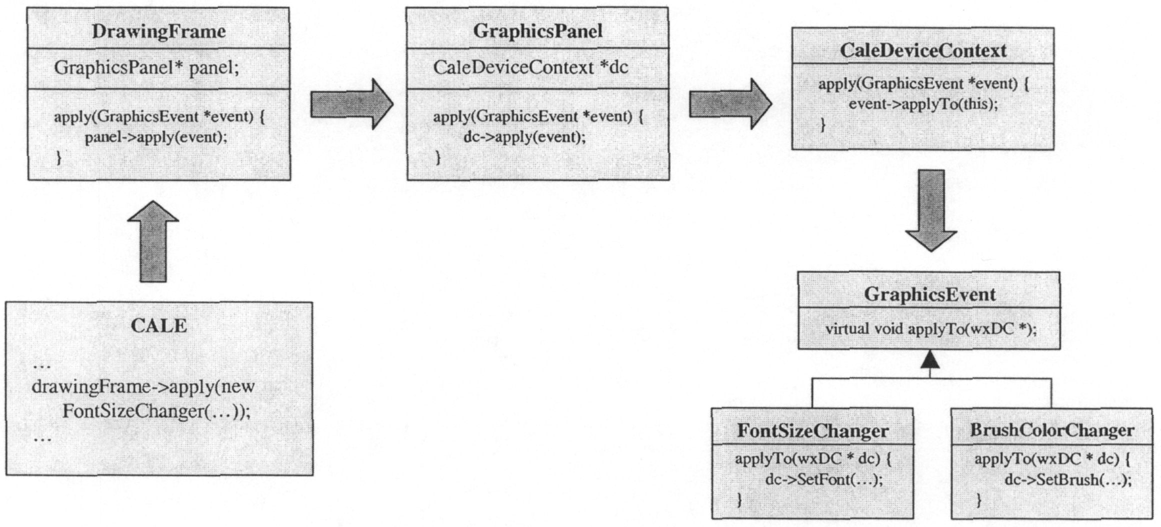

DrawingFrame
GraphicsPanel* panel;
apply(GraphicsEvent *event) { panel->apply(event); }
GraphicsPanel
CaleDeviceContext *dc
apply(GraphicsEvent *event) { dc->apply(event); }
CaleDeviceContext
apply(GraphicsEvent *event) { event->applyTo(this); }
CALE
... drawingFrame->apply(new FontSizeChanger(...)); ...
GraphicsEvent
virtual void applyTo(wxDC *);
FontSizeChanger
applyTo(wxDC * dc) { dc->SetFont(...); }
BrushColorChanger
applyTo(wxDC * dc) { dc->SetBrush(...); }

Figure 1. Function calling sequence when drawing is performed.

Console Implementation Details

Keeping the Dialogs in Sync with Command Line

Another consequence of the command line interface and graphics window being in separate threads is that we need to keep the dialog boxes synchronized with what is going on at the command line. For example, if we have the pick dialog box open and the user types "addvar den," the pick dialog needs to be able to find out that "den" was added to the list of pick variables. A similar situation occurs in places like the plot1d dialog where there are checkboxes showing whether different flags have been set such as ifxlog. Anywhere that a dialog displays a variable that can be manipulated via the command line, the dialog needs to be able to know when the variable's value has changed so that it can update itself accordingly.

The way that this is accomplished is by using a design pattern called the observer pattern. In the observer pattern, the first thing that you typically need is an abstract base class that declares pure virtual functions that will be implemented by listeners to allow them to be notified when some type of event happens. For example, one such pure virtual function might be variableValueChanged(char* varName, int newValue). This base class is typically named something like MouseMotionListener or VariableChangedListener. Then, you need write some classes that have this listener class as a base class and that are therefore required to implement the virtual functions. Somewhere else, usually in another class, you then have a list of listeners which have registered themselves as being interested in a certain type of event by adding themselves to the list. When an interesting event happens in the class, say the changing of the value of a variable, the class goes through its list of listeners and calls the virtual function on each listener.

In CALE, the observer pattern is implemented slightly differently. CaleApp, the main application class, contains two lists of listeners: one for when variable values change and one for when pick events occur. In CALE, when interesting events happen, the function in the original CALE code gets a pointer to the CaleApp class and then tells it (or, more specifically, one of its listener lists) that an interesting event has happened. That then causes all of the registered listeners to be notified. This notification process is illustrated by the following diagram.

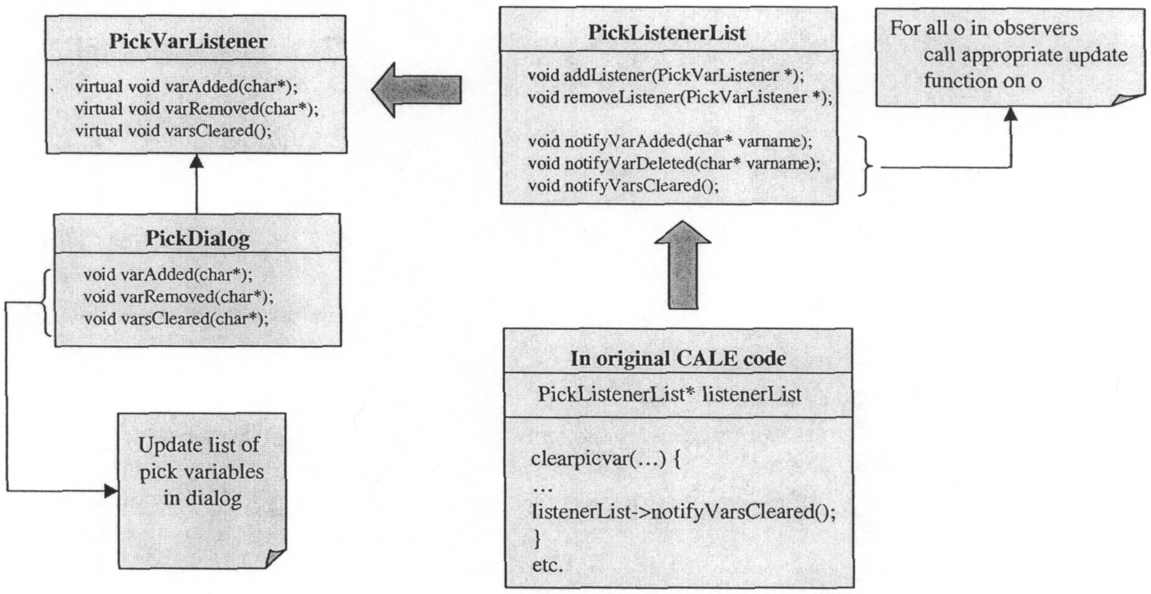

PickVarListener
virtual void varAdded(char*);
virtual void varRemoved(char*);
virtual void varsCleared();
PickListenerList
void addListener(PickVarListener *);
void removeListener(PickVarListener *);
void notifyVarAdded(char* varname);
void notifyVarDeleted(char* varname);
void notifyVarsCleared();
For all o in observers call appropriate update function on o
PickDialog
void varAdded(char*);
void varRemoved(char*);
void varsCleared(char*);
Update list of pick variables in dialog
In original CALE code
PickListenerList* listenerList
clearpicvar(...) {
...
listenerList->notifyVarsCleared();
}
etc.

Figure 2. Function calling sequence when the pick variables change.

Calling CALE Functions from Dialogs

Strange as it may seem, there are a couple issues associated with calling CALE functions from within Dialogs. The main issue occurs because when you call CALE functions from a dialog they are run in the graphics thread. This is a problem because some of the CALE functions such as zoom and pick assume that they are not being called from the graphics thread. These functions essentially put themselves to sleep until the user has clicked on the drawing frame or done something similar. The graphics thread is then unable to respond to button clicks and such since it is asleep. In order to call these types of functions, a separate thread is created and the function is called inside that thread. The other issue associated with calling CALE functions from within Dialogs is synchronization, and that was discussed in the Synchronization section.

Uniform Handling of Different Command Types using Actions

Within the dialogs that make up the Console, there are a couple of list boxes where each item in the list box is associated with some sort of a command. Among these commands, there are many different command types. For example, some commands take arguments and some don't. Some commands simply need to change the value of a variable and refresh the display whereas others need to pass a command to CALE's parser. Furthermore, some commands must be executed in a separate thread. In order to accommodate the different requirements of the different commands, the different things that a command can do have been abstracted into different action classes. For example, there is a class called `FunctionCallingAction` that calls a function, a class called `ValueSettingAction` that sets the value of some variable, and a `RefreshAction` class that refreshes the display if the AutoRefresh checkbox is selected. Each of these classes is required to implement the functions `onSelected()`, `onDeselected()`, and `onArgumentsEntered(const char*)`. When an instance of the class is created you specify the details about what should happen and whether the action should occur when the items is selected, when it is deselected, when arguments are entered, or some combination of those three.

Gluing all of these actions together is a class called `ActionList`, which is essentially a list of actions. Actions are added to an `ActionList` by the dialog and then, at the appropriate time, the actions are executed in the order they were added. The `ActionList` provides support for executing its commands in a separate thread and also allows for `SkippingActions` that can cause certain actions to be skipped if a certain condition is true.

The way that this is implemented for the listboxes mentioned earlier is that the dialog has an array of `ActionLists`. In other words, each item in the listbox has an `ActionList` associated with it. When the item is selected, all of the `onSelected()` functions are called, and when the item is deselected, all of the `onDeselected()` functions are called. The way that this works is illustrated by the following diagram.

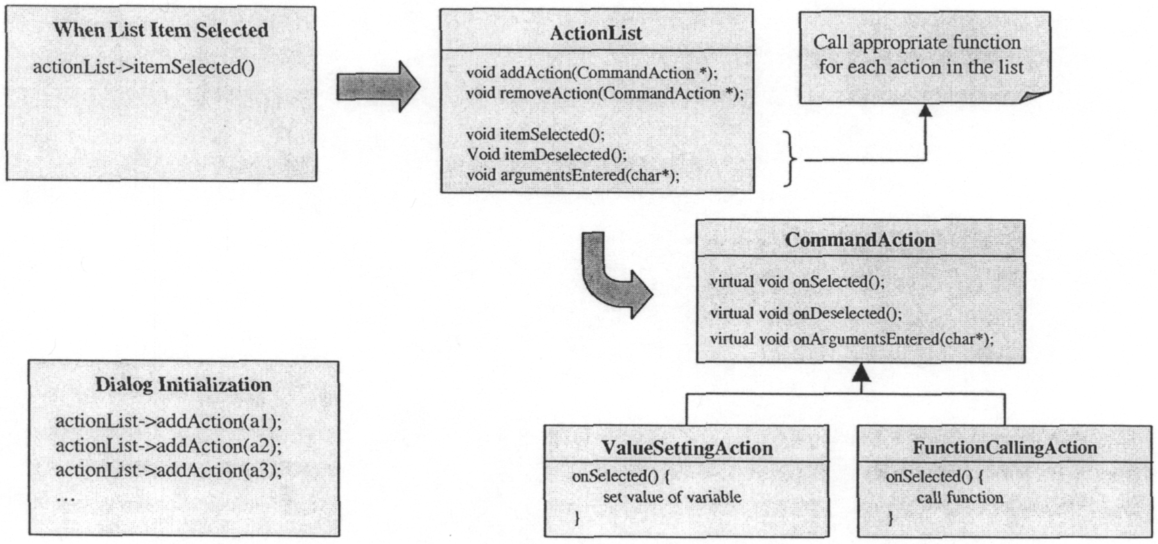

When List Item Selected
actionList->itemSelected()
ActionList
void addAction(CommandAction *);
void removeAction(CommandAction *);
void itemSelected();
Void itemDeselected();
void argumentsEntered(char*);
Call appropriate function for each action in the list
CommandAction
virtual void onSelected();
virtual void onDeselected();
virtual void onArgumentsEntered(char*);
Dialog Initialization
actionList->addAction(a1);
actionList->addAction(a2);
actionList->addAction(a3);
...
ValueSettingAction
onSelected() {
set value of variable
}
FunctionCallingAction
onSelected() {
call function
}

Figure 3. Function calling sequence when a list item is selected.

Reusing Code with Templates

Templates are a very powerful feature of C++ that allow the creation of functions that are essentially parameterized by the types of parameters that are used in them. When these types are classes, templates can actually be used as a substitute to true polymorphism. When you are using true polymorphism, you have a base class with at least one virtual function and then have subclasses that override those functions. Then, you can use the subclass objects as if they were objects of the base class type. When a virtual function is called on the base class type, a virtual function table (vtable) is used to determine what subclass function to call, and then this function is invoked.

Something very similar to this can be accomplished using templated classes. When using templates, there is no need for a base class or for virtual functions. When writing a function, you simply use the parameterized type as if it was a base class type with certain virtual functions. Then, any class you use as the parameter when the templated class is instantiated is statically forced to implement those functions. If the functions are not implemented, then the program will not compile. What the function does in the templated class now depends on what the templated type is. This is sometimes referred to as static or parametric polymorphism and is frequently used to accomplish polymorphism without having the overhead of using virtual function tables.

There are a number of places where templates are used in CALE. Static polymorphism is used in the `FindPanel` class. What the `FindPanel` basically does is go through some sort of a list and cycle through the entries that match a certain string. This list can be either a `wxComboBox` or a `wxListBox`, which are classes that are implemented by wxWindows. Using static polymorphism allows functions with the same name to be called on different classes without having to create a common base class for the two list types. In this case, the classes wxComboBox and wxListBox both have the methods `GetString(int)`, `SetSelection(int)`, and `GetCount()`.

Probably the most notable use of templates is in the different thread classes. These thread classes allow functions with a varying number of parameters to be called from within a separate thread. The thread dies as soon as the function is done executing. To accomplish this, one thread class was created for each different number of parameters. The type of the function and types of the parameters are template arguments. This allows the same thread classes to be used with an arbitrary number of different parameter types.

Future Work

There is still a bit of work that needs to be done on the wxWindows version of CALE as of this writing. In the Synchronization section it was mentioned that all of the functions that were called by the Console dialogs needed to be protected by a mutex. This is because in the unlikely event that a CALE function is being executed in the Console thread at the same time that a CALE function is being executed in the graphics thread, the data structures could become corrupted if the two threads try to modify them at the same time.

In addition, the current method of repainting uses a great deal of memory. Essentially, every time a graphics event occurs, it is stored in a list. When repaint events occur, the graphics events in the list are then executed again. The list is cleared when clear events are processed. A more memory efficient way to repaint would be to just call `fm()`. Unfortunately, when that option was tried, a segmentation fault occurred during repaint events and the program crashed. It would be worthwhile to look into the cause of this. It may be related to synchronization. Another possibility is to use the `Blit(...)` function in the `wxDC` class (which `CaleDeviceContext` ultimately inherits from) to efficiently copy the display window onto some other `wxDC` after each drawing event. The contents of that device context could then be blitted back to the window whenever a repaint event occurs.

However, saving the drawing events in a list does have its benefits, although none of them are being exploited at the moment. The events stored in the list could easily be applied to, for example, a `wxPrinterDC` in order to print the current contents of the graphics window or to a `wxMetaFileDC` to create a metafile. It may be possible to accomplish these things using the blitting approach, though.

Currently `addvar`, the function that adds pick variables, allows duplicate pick variables to be added. That is probably not a good idea.

Additionally, the `Plot1DDialog` currently allows any curves to be highlighted. It would probably be better to just show the curves which have been created and then create a `CurveModificationListener` which would allow the `Plot1DDialog` to find out when curves are added or deleted and to update the list accordingly.

Porting CALE to wxWindows

Jeffrey Hagelberg
August 13, 2003

Outline

- Project Overview
- Overall Strategy
- Graphics Abstraction
- Static Polymorphism
- Observer Pattern
- Results

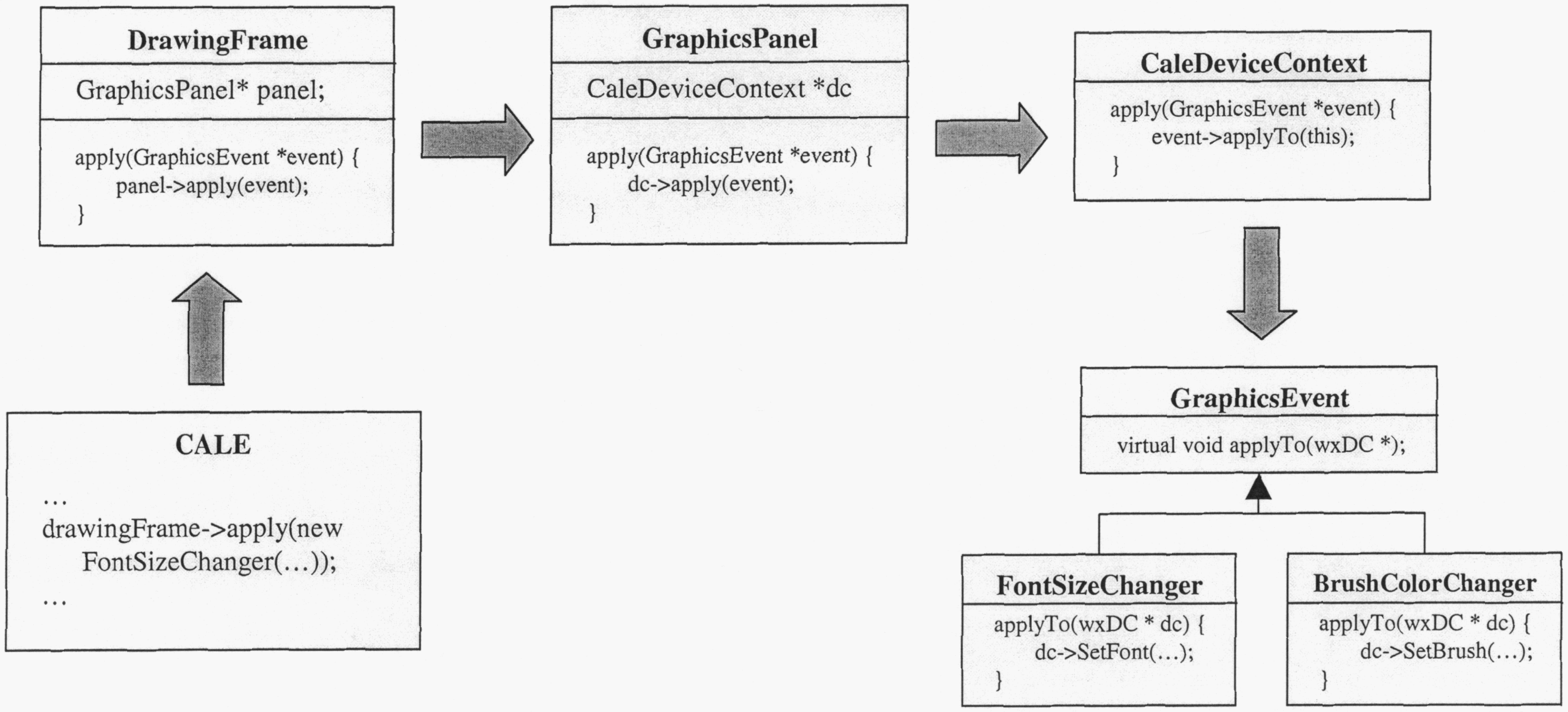
Project Overview

- Goals
  - Make CALE run on Windows
  - Platform Independence
  - Preserve existing functionality
  - Modernize user interface

Overall Strategy

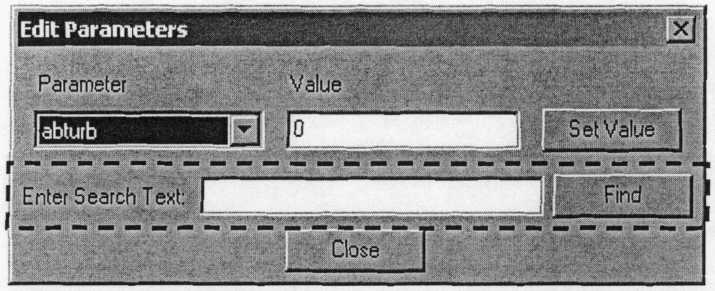
- Use wxWindows
- MinGW (Minimalist GNU for Windows)
- Compile as C++
- Conditional Compilation Directives (#ifdefs)
- Separate console thread
- Encapsulate the graphics

Graphics Abstraction

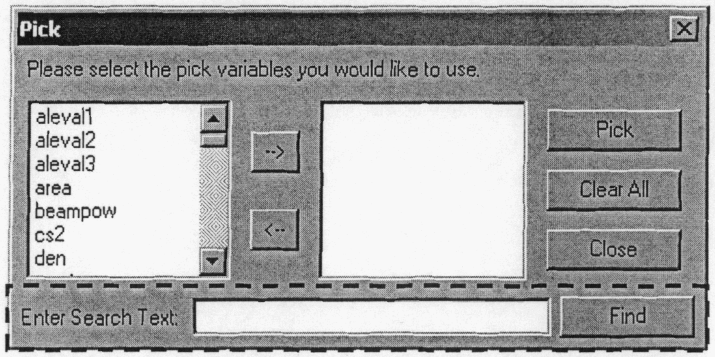


Static Polymorphism

Can you see what all of these dialogs have in common?

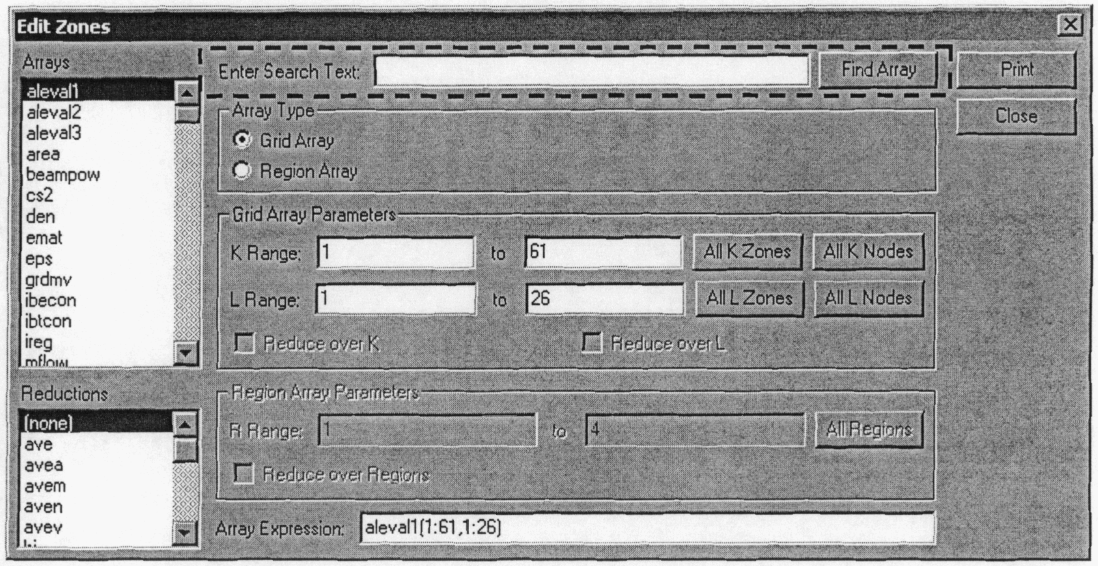


Parameter: abturb
Value: 0



Please select the pick variables you would like to use.

aleval1
aleval2
aleval3
area
beampow
cs2
den



Arrays: aleval1, aleval2, aleval3, area, beampow, cs2, den, emat, eps, grdmv, ibecon, ibtcon, ireg, mflow

Reductions: (none), ave, avea, avem, aven, avev

Array Type: [✓ Grid Array] Region Array

K Range: 1 to 61
L Range: 1 to 26
R Range: 1 to 4

Array Expression: aleval1(1:61,1:26)

Static Polymorphism

FindPanel.h

```
template<class listType>
class FindPanel : public wxPanel {
 public:
  FindPanel(wxWindow *parent,
            listType* list,
            wxString findButtonText = wxString("Find"));
  ~FindPanel();
  void onFindClicked(wxCommandEvent &event);
private:
  listType* m_list;
};
```

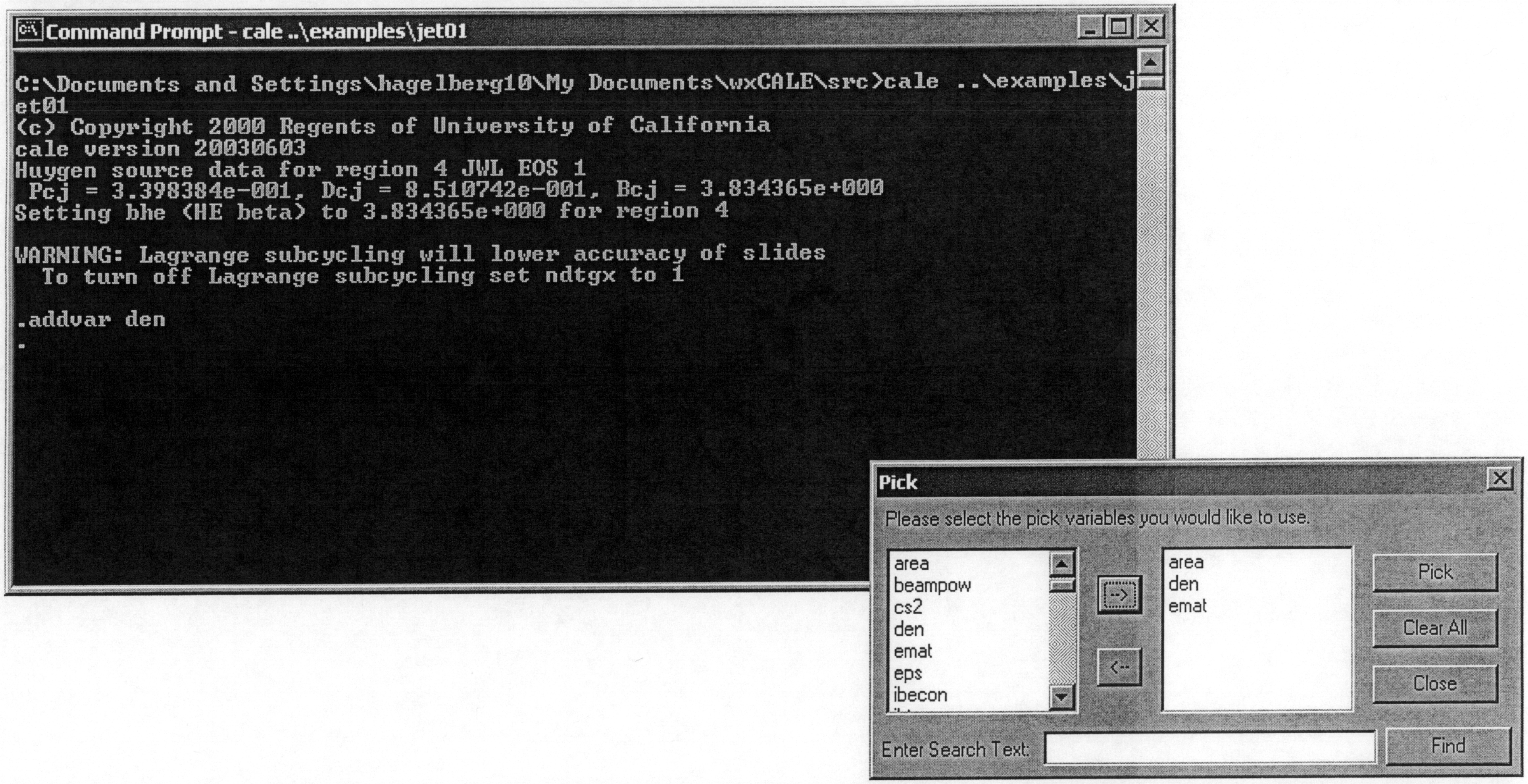
Static Polymorphism

FindPanel.cpp

```
template<class listType>
void FindPanel<listType>::onFindClicked(wxCommandEvent &event) {
  wxString searchText = m_findTC->GetValue();
  int startIndex = (m_lastIndex + 1) % m_list->GetCount();

  int curIndex = startIndex;
  do {
        ...
    wxString curVarName(m_list->GetString(curIndex));
    curIndex = (curIndex + 1) % m_list->GetCount();
        ...
  } while(curIndex != startIndex); //only go around once
}
```

Observer Pattern

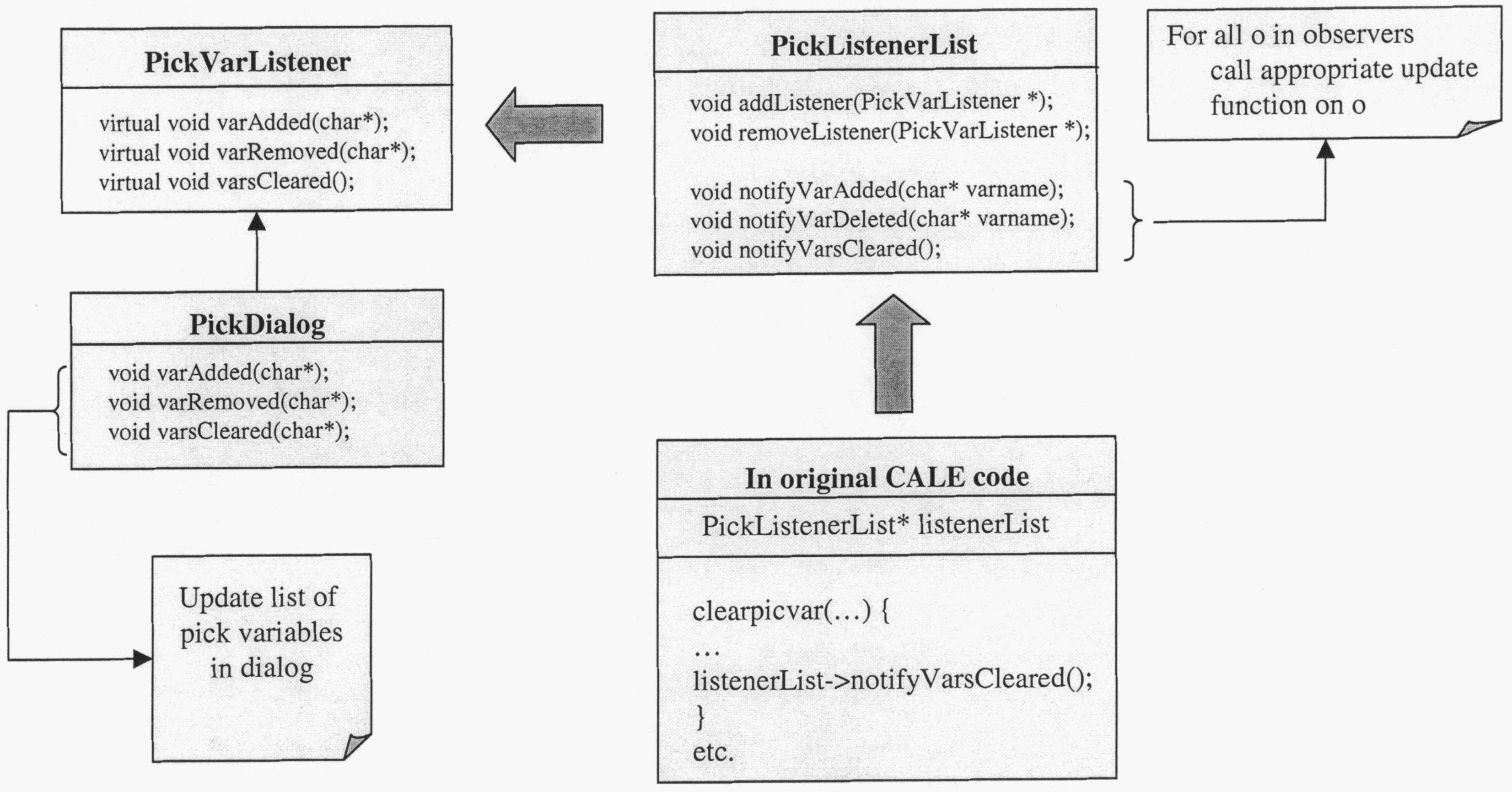


```
C:\Documents and Settings\hagelberg10\My Documents\wxCALE\src>cale ..\examples\j
et01
(c) Copyright 2000 Regents of University of California
cale version 20030603
Huygen source data for region 4 JWL EOS 1
 Pcj = 3.398384e-001, Dcj = 8.510742e-001, Bcj = 3.834365e+000
Setting bhe (HE beta) to 3.834365e+000 for region 4

WARNING: Lagrange subcycling will lower accuracy of slides
  To turn off Lagrange subcycling set ndtgx to 1

.addvar den
-
```

Observer Pattern



PickVarListener

```
virtual void varAdded(char*);
virtual void varRemoved(char*);
virtual void varsCleared();
```

PickDialog

```
void varAdded(char*);
void varRemoved(char*);
void varsCleared(char*);
```

Update list of pick variables in dialog

PickListenerList

```
void addListener(PickVarListener *);
void removeListener(PickVarListener *);

void notifyVarAdded(char* varname);
void notifyVarDeleted(char* varname);
void notifyVarsCleared();
```

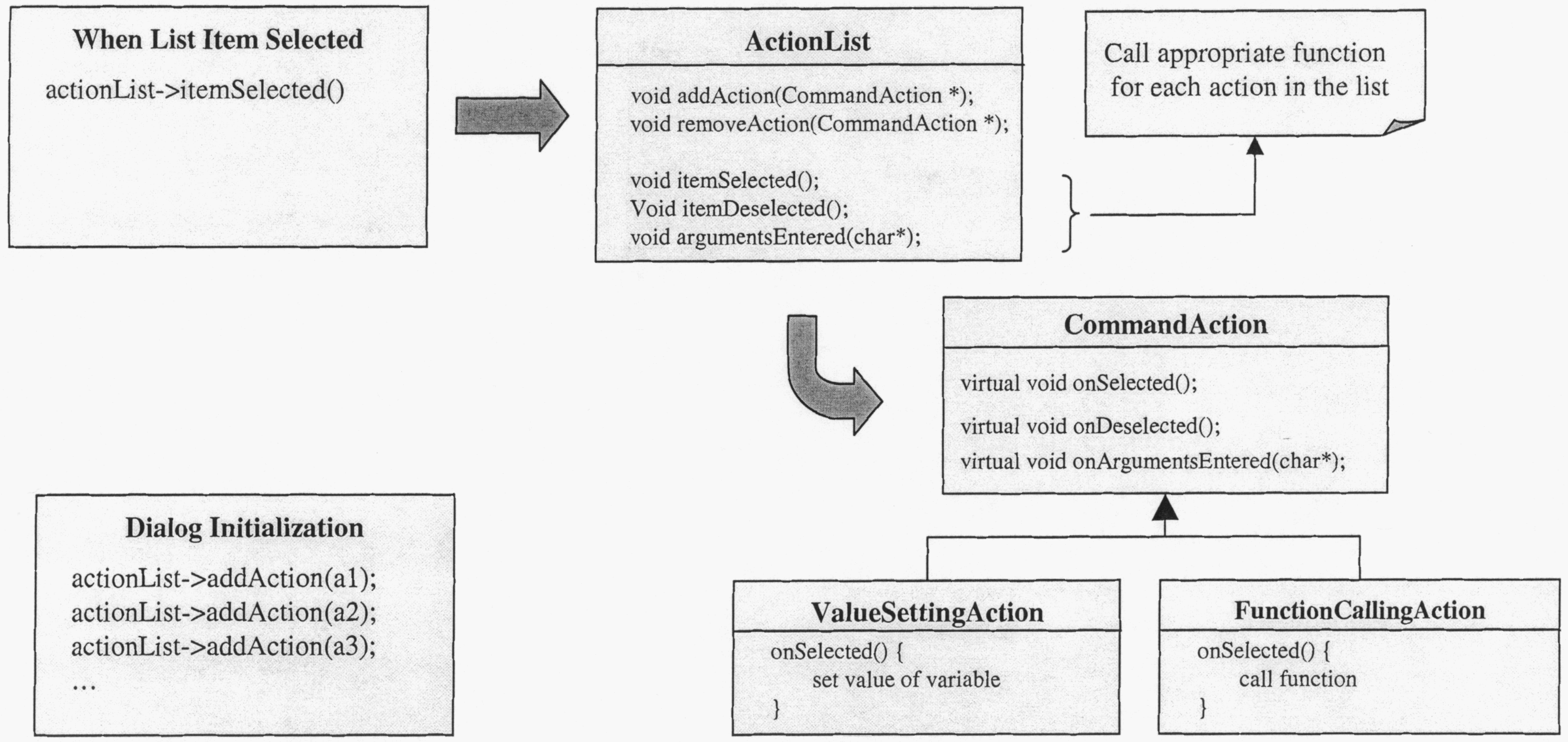
For all o in observers
call appropriate update function on o

In original CALE code

PickListenerList* listenerList

```
clearpicvar(...) {
...
listenerList->notifyVarsCleared();
}
etc.
```

Command Pattern



When List Item Selected

```
actionList->itemSelected()
```

ActionList

```
void addAction(CommandAction *);
void removeAction(CommandAction *);

void itemSelected();
Void itemDeselected();
void argumentsEntered(char*);
```

Call appropriate function for each action in the list

CommandAction

```
virtual void onSelected();
virtual void onDeselected();
virtual void onArgumentsEntered(char*);
```

Dialog Initialization

```
actionList->addAction(a1);
actionList->addAction(a2);
actionList->addAction(a3);
...
```

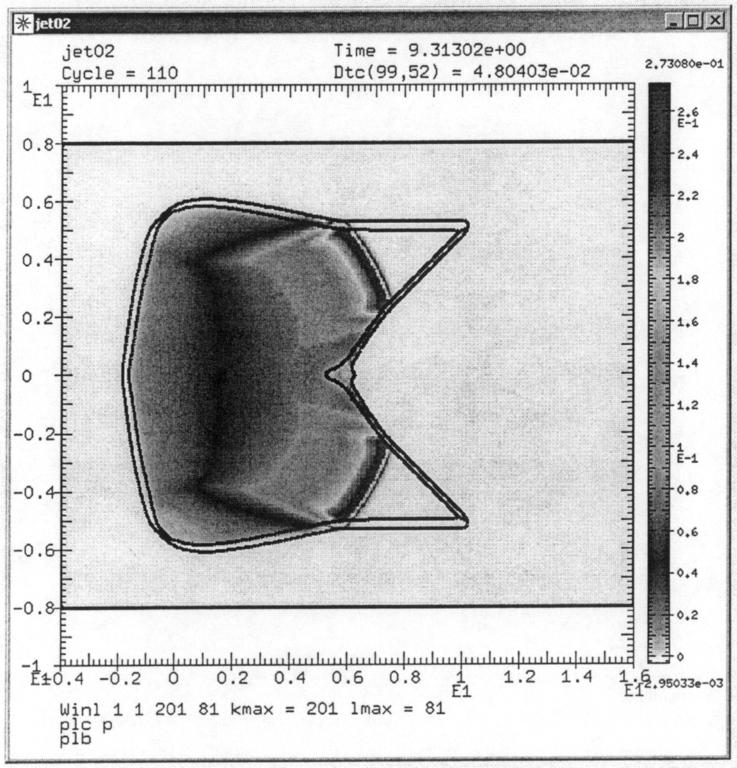
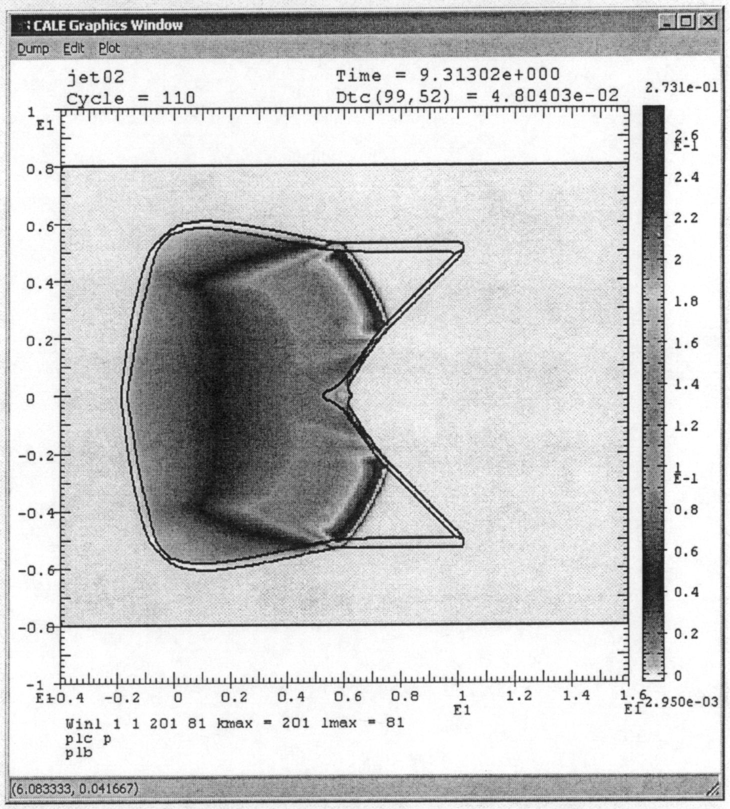
ValueSettingAction

```
onSelected() {
    set value of variable
}
```

FunctionCallingAction

```
onSelected() {
    call function
}
```

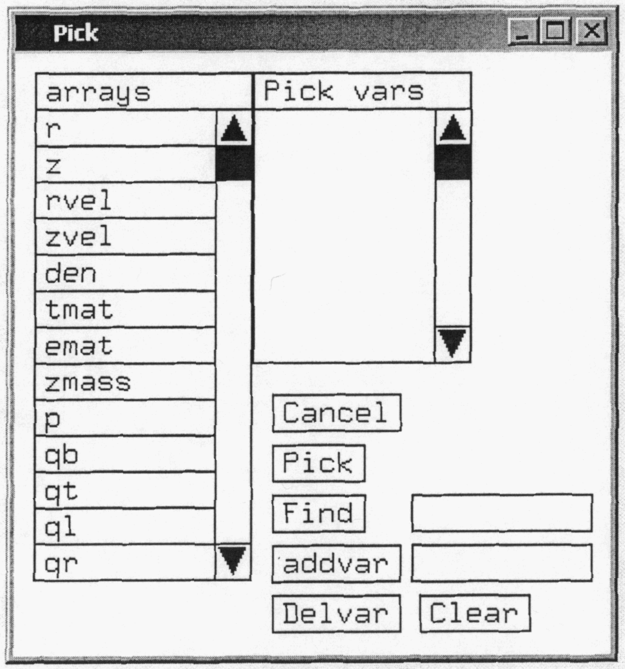
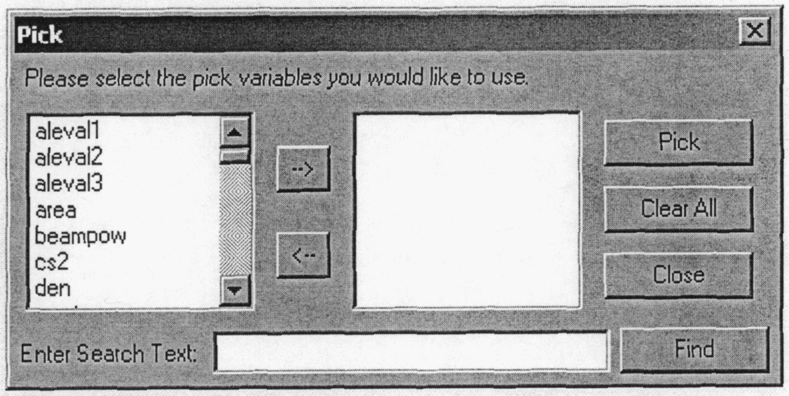
Results

Original Version	wxWindows Version
	

```
Winl 1 1 201 81 kmax = 201 lmax = 81
plc p
plb
```

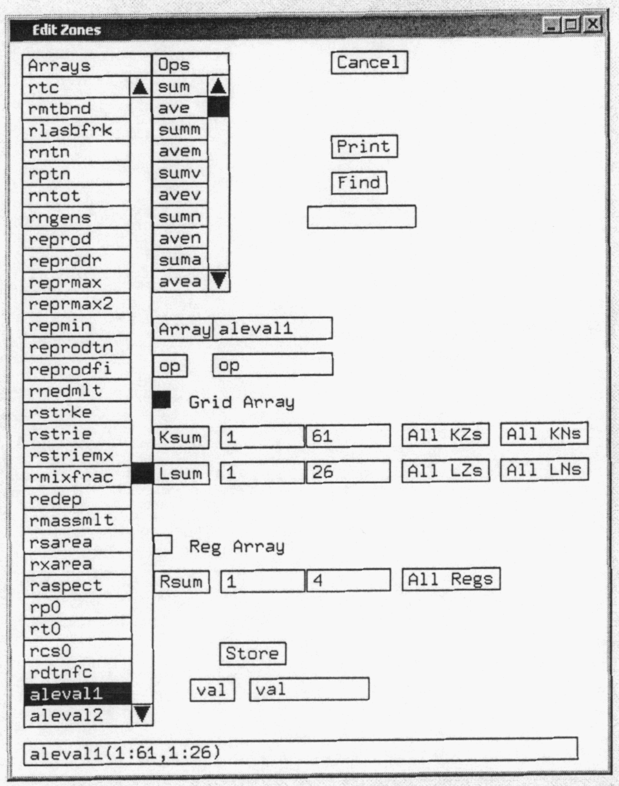
```
Winl 1 1 201 81 kmax = 201 lmax = 81
plc p
plb
```

Results

Original Version	wxWindows Version
	

Results

Original Version	wxWindows Version



Arrays: rtc, rmtbnd, rlasbfrk, rntn, rptn, rntot, rngens, reprod, reprodr, reprmax, reprmax2, repmin, reprodtn, reprodfi, rnedmlt, rstrke, rstrie, rstriemx, rmixfrac, redep, rmassmlt, rsarea, rxarea, raspect, rp0, rt0, rcs0, rdtnfc, aleval1, aleval2

Ops: sum, ave, summ, avem, sumv, avev, sumn, aven, suma, avea

Array: aleval1

op: op

[✓ Grid Array]

Ksum: 1 – 61

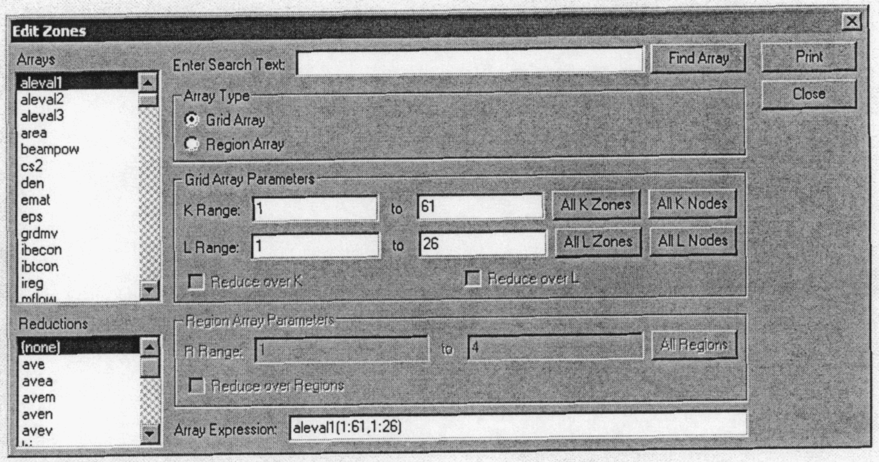
Lsum: 1 – 26

Reg Array [ ]

Rsum: 1 – 4

val: val

aleval1(1:61,1:26)



Arrays: aleval1, aleval2, aleval3, area, beampow, cs2, den, emat, eps, grdmv, ibecon, ibtcon, ireg, mflow

Enter Search Text:

Array Type: [✓ Grid Array] Region Array

Grid Array Parameters
K Range: 1 to 61
L Range: 1 to 26
[ ] Reduce over K [ ] Reduce over L

Reductions: (none), ave, avea, avem, aven, avev

Region Array Parameters
R Range: 1 to 4
[ ] Reduce over Regions

Array Expression: aleval1(1:61,1:26)

Any Questions?